1 Administrivia

- Pset2 Beta due Thursday March 1
- Pset2 Final due Friday March 9

2 Recursive Data Types

See slides of Lecture 06 and Additional Note on Recursive Data Types on Stellar.

To implement recursive data types, we use an "interface". An interface is a reference type, like a class, that can contain only constants, method signatures, and nested types. There are no method bodies. Interfaces cannot be instantiated; they can only be implemented by classes or extended by other interfaces. To use an interface, we write a class that implements the interface. When an instantiable class implements an interface, it provides a method body for each of the methods declared in the interface (See Java docs for "interfaces").

```
public interface ImList<E> {
    public ImList<E> cons(E e);
    public E first();
    public ImList<E> rest();
}
```

Then we define two classes that "implement" the interface. Empty represents the result of the empty operation (an empty list), and Cons represents the result of a cons operation (an element glued together with another list):

```
public class Empty<E> implements ImList<E> {
    public Empty() { }
    public ImList<E> cons(E e) { return new Cons<E>(e, this); }
    public E first() { throw new UnsupportedOperationException(); }
    public ImList<E> rest() { throw new UnsupportedOperationException(); }
}

public class Cons<E> implements ImList<E> {
    private E e;
    private ImList<E> rest;
    public Cons(E e, ImList<E> rest) {
        this.e = e;
        this.rest = rest;
    }
    public ImList<E> cons(E e) {
        return new Cons<E>(e, this);
    }
    public E first() {
        return e;
    }
    public ImList<E> rest() {
        return rest;
    }
}
```

Some relevant concepts (e.g., "rep invariant") are to be covered in the near future under ADTs; see below Section 4 for a brief description for the time being.

We will practice with "Expressions" and "Binary Tree" recursive data types in this session.
3 Satisfiability (SAT)

3.1 Problem Definition
The satisfiability problem, or SAT, is the problem of finding, for a given boolean expression consisting of one or more variables, an assignment that causes the expression to evaluate to true. For example:

\[ x + y + z' \]

has for one of its solutions the assignment \((x, y, z) = (T, T, T)\).

In most cases, we will be studying a special version of SAT known as 3-SAT, which is actually provably the same problem as the general case. The difference is that we require our equations to be in conjunctive normal form, which looks like:

\[(x_1 + x_2 + x_3) \cdot (x'_1 + x_2 + x_3) \cdot (x_2 + x'_4 + x'_6) \cdot \ldots\]

Each clause in this case is the boolean disjunction (OR) of three literals, which can either be a variable of the negation of a variable. The formula itself is then defined as the conjunction (AND) of all of the clauses.

3.2 DPLL
The DPLL, or Davis-Putnam-Logemann-Loveland, algorithm is a simple backtracking-based routine for solving instances of 3-SAT. It has a few basic steps for solving a given formula, \(\phi\). The pseudocode for DPLL(\(\phi\)) looks like:

1. If \(\phi\) is consistent and completely assigned, return true
2. If any clause in \(\phi\) is empty, return false
3. If there are any unit clauses (clauses with only one literal), apply unit propagation to them such that for the single variable in the clause, \(x\):
   - \(x\) is replaced with \(T\) throughout \(\phi\)
   - \(x'\) is replaced with \(F\) throughout \(\phi\)
4. Pick the next unassigned literal, \(y\), in \(\phi\)
   - Arbitrarily assign either \(T\) or \(F\) to \(y\) and recurse.
   - If that fails, try the other option.

4 Abstract Data Types
An abstract data type (ADT) is just an abstract description of how instances of a certain type behave defined through the methods that type exposes. This includes, but is not restricted to:

- Preconditions — any conditions that need to be met before the behavior of the method is defined. This includes any arguments the method might take.
- Postconditions — any side-effects that calling this method might have. This includes returning values.

An example specification in this style might look like:

```java
static int find (int [] a, int val)
    requires: val occurs exactly once in a
    effects: returns result such that a[result] = val
```
4.1 Representation Exposure

Representation exposure occurs when implementation decisions leak incorrect behavior through an abstract interface. This is bad for two major reasons: it can potentially lead to bugs or faulty behavior, and it can inadvertently couple the implementation of an interface to the client of an interface. Some common things that can lead to representation exposure:

- Inconsistent state (e.g. `size` variable not being updated)
- Exposing internal state (e.g. returning a mutable data structure)
- Poorly defined interfaces (e.g. Page/Weather cache)

4.2 Representation Invariants

A representation invariant (or "rep invariant" for short) is a formal constraint that specifies when an instance of an abstract data type is well-formed. Why do we bother with representation invariants? Because they:

- Make modular reasoning possible. Without the rep invariant documented, you might have to read all the methods to understand what’s going on before you can confidently add a new method.
- Help catch errors. By implementing the invariant as a runtime assertion, you can find bugs that are hard to track down by other means.

Representation invariants are typically expressed as a function that maps an instance of an abstract data type (i.e. an object) to a boolean value that is true if and only if the object is valid. In order to be able to ensure validity of all instances of a type, rep invariants need to be checked when an instance is:

- Created, most of the time in a constructor
- Changed, through methods that mutate its state

By guaranteeing the validity of an instance in these two places, we can essentially inductively guarantee that the instance will always be valid, as long as we don’t commit the sin of representation exposure.

```
public TicTacToe {
    private static final CHECK_REP = true;
    private final Move[][] board = new Board[3][3];

    public TicTacToe(){
        ...
        if (CHECK_REP) checkRep();
    }

    public void doMove(int r, int c, Move player) {
        ...
        if (CHECK_REP) checkRep();
    }

    public Move getVal(int r, int c) { ... }

    private void checkRep(){
        int countX = ...; int countO = ...;
        assert Math.abs(countX - countO) <= 1 :
            "Invalid number of X and Os on board. #X - #O = " + countX - countO;
        int numWinners = ...;
        assert numWinners <= 1 : "More than one winner on board";
    }
}
```

Listing 3: Example of `checkRep()` using Tic Tac Toe